Periodica Polytechnica
Electrical Engineering
and Computer Science

61(2), pp. 132-150, 2017
https://doi.org/10.3311/PPee.9755

RESEARCH ARTICLE

Model Checking-based Software-FMEA: Assessment of Fault Tolerance and Error Detection Mechanisms

Vince Molnár[1,2*], István Majzik[1]

Received 17 July 2016; accepted after revision 27 November 2016

Abstract

Failure Mode and Effects Analysis (FMEA) is a systematic technique to explore the possible failure modes of individual components or subsystems and determine their potential effects at the system level. Applications of FMEA are common in case of hardware and communication failures, but analyzing software failures (SW-FMEA) poses a number of challenges. Failures may originate in permanent software faults commonly called bugs, and their effects can be very subtle and hard to predict, due to the complex nature of programs. Therefore, a behavior-based automatic method to analyze the potential effects of different types of bugs is desirable. Such a method could be used to automatically build an FMEA report about the fault effects, or to evaluate different failure mitigation and detection techniques. This paper follows the latter direction, demonstrating the use of a model checking-based automated SW-FMEA approach to evaluate error detection and fault tolerance mechanisms, demonstrated on a case study inspired by safety-critical embedded operating systems.

Keywords

Failure Mode and Effects Analysis, SW-FMEA, model checking, fault tolerance, error detector

[1] Department of Measurement and Information Systems,
Faculty of Electrical Engineering and Informatics,
Budapest University of Technology and Economics,
H-1521 Budapest, P.O.B. 91, Hungary

[2] MTA-BME Lendület Cyber-Physical Systems Research Group,
Budapest, Hungary

* Corresponding author, e-mail: molnarv@mit.bme.hu

1 Introduction

The risk of failure is one of the main concerns of safety-critical systems. Certification requires the systematic analysis of potential failures, their causes and effects, and the evaluation of risk mitigation techniques used to reduce the chance and the severity of system-level failures.

One of the first systematic techniques for failure analysis was Failure Mode and Effect Analysis (FMEA) [2]. FMEA is often the first step in reliability analysis, as it collects the potential failure modes of subsystems, their root causes and their effects on the whole system. Together with criticality analysis (often treated as part of FMEA, sometimes emphasized by the term FMECA), the output of FMEA serves as the basis of other qualitative and quantitative analyses, as well as design decisions regarding risk mitigation techniques.

FMEA is usually applied at the hardware and communication level, where it requires a qualified analyst to collect postulated component failures and identify their effects on other components and the system level. In case of software (SW-FMEA), failure modes originate in different types of programming faults, commonly referred to as bugs. Due to the complex nature of software and the many types of potential bugs, it is much harder to collect failure modes and deduce their potential effects, so an automated mechanism is desired.

This paper presents a way of automated SW-FMEA with the use of executable software models. Assuming a set of predefined fault types (programming faults) and a specification of safe behavior at the system level, the proposed approach applies model checking to systematically generate execution traces leading from fault activations to states that violate the specification of safe behavior (system-level failures). These traces can be used to understand and demonstrate fault propagation through the system and also as test sequences to reveal actual faults in the final product.

Our approach improves existing model checking-based SW-FMEA by optimizing fault activations and using monitor components instead of a formal specification. The optimization helps in scaling the solution to apply it on more complex systems and situations, whereas using a monitor instead of formal

specifications to describe correct and incorrect behaviors is closer to the way engineers think about systems, facilitating correctness and productivity.

In addition to automated SW-FMEA, the proposed approach can be used to evaluate the efficiency of fault tolerance and error detection mechanisms. In traditional FMEA, the enumeration of possible faults and their effects is followed by the definition of detection and mitigation mechanisms targeting the given fault (this section of the FMEA report is commonly called *disposition*) [18]. In case of off-the-shelf components (COTS), these mechanisms are often compiled into a wrapper to reduce the risk of using the component [3]. In such a setting, it is equally important to demonstrate that these mechanisms are indeed capable of detecting and isolating the faults they are meant to handle. Fault tolerance mechanisms should mask as many faults as possible (reducing the number of fault activations that can lead to a system-level failure), while error detectors should catch the propagating error on as many traces as possible. To the best of our knowledge, no other work addresses the problem of evaluating such auxiliary components in a model-based FMEA setting.

Throughout the paper, a running example will be used to demonstrate the introduced concepts and to serve as a detailed walk-through for readers to help in implementing the approach for their own systems. We also employ the evaluation of error detectors on an industrial case study, using a model of the OSEK API specification [1], which is a commonly used interface specification for embedded operating systems.

This paper is an extended version of [19]. In addition to the ideas outlined in the conference paper, this version includes a detailed case study and running example to explain, refine and clarify the proposed approach, providing various alternatives to implement it in different projects.

The paper is structured as follows. Section 2 introduces the key concepts of FMEA and model checking. Section 3 describes the running example, then a framework for model checking-based FMEA is outlined in Section 4. Applications of the approach are discussed in Section 5, while Section 6 presents an industrial case study. Section 7 provides the concluding remarks and our directions for future work.

2 Background

This section summarizes the main idea of FMEA and in particular SW-FMEA, as well as model checking that is the basis of the approach presented in Section 4.

2.1 Failure Mode and Effects Analysis

FMEA involves *1)* the enumeration of potential failure modes of subsystems, *2)* an inductive reasoning of their effects (or costs) on different levels of the whole system (called *error propagation*), and *3)* often the deductive analysis of their root causes. The analysis is usually based on a model or specification of a component, as well as historical data about faults and failures (in case of software, the typical types of bugs) and experience with similar components. The result is recorded in an FMEA spreadsheet [18]. Failure modes are then categorized based on criticality, representing the level of chance and the severity of potential consequences. Criticality can prioritize failures, and based on the discovered causes and effects, fault-tolerance or error detection mechanisms can be designed to mask faults or detect errors to ensure fail-safe operation of the system.

There are three main concepts related to error propagation [5]. A *failure* is an incorrect system function, i.e., an observable incorrect state. An *error* is a latent incorrect state that has no observable effects yet. Finally, a *fault* is the cause of a failure, which can be either some kind of defect (physical or design) or the failure of a related subsystem.

During error propagation, an *activated fault* can cause an error, which will turn into a failure once it becomes observable (e.g., by crossing an interface). FMEAs are usually performed on many levels during the design of a system, so a failure of a component is often a fault on another level. FMEA usually assumes that only a single failure mode exists at a time. Examples for safety analysis techniques analyzing error propagation include the HIPS-HOPS [21] and FPTC [25] techniques, where components are analyzed in isolation from the system to model their failure behavior, enabling the estimation of fault propagation in the whole system.

Fault-tolerance of the final system is often demonstrated by hardware fault injection and intentional corruption of inputs or memory [10]. This process can be regarded as the testing of the mechanisms designed during the FMEA process, but can also serve as an automated way to discover the effects of different faults.

2.1.1 Software FMEA

Fault injection with software was initially introduced to emulate hardware faults, an approach commonly referred to as Software Implemented Hardware Fault Injection [10]. However, when performing FMEA on software components, it is interesting to consider failure modes caused by programming (or configuration) faults. The challenge of analyzing these is twofold. First, it is very hard to come up with a realistic set of programming faults (called a *fault model*). The source of bugs is almost always a human, and the most typical faults highly depend on the programming language and the domain as well. Various studies tried to identify the most common types of software faults (e.g. [13]). Studies also revealed that it is very hard to accurately simulate real-life faults with fault injection [17]. Constructing a realistic fault model is even harder in case only a design model is available. The second challenge lies in the difficulty of tracking the effects of a bug as it evolves in a complex system.

This paper focuses on the challenge of analyzing the effects of programming faults. The problem of designing proper fault models is not discussed here, we refer to the fact that it is also an important challenge in the field of *mutation-based testing*. For an extensive overview of mutation-based testing and fault models, the reader should refer to [16]. More studies about the various aspects of software fault injection and fault models can be found in [12, 13, 17, 24].

Most of the previous approaches to SW-FMEA build on software testing (e.g., [22]), injecting faults directly into the program code and running a set of tests to see any possible global effects. There are mixed approaches, where experiments are combined with analytical models to measure fault-tolerance (e.g., [4]). Using fault injection to compute the risk associated with different (often off-the-shelf) components is thoroughly discussed in [20]. These methods are mostly useful once an implementation already exists. However, FMEA and the design of fault-tolerance mechanisms should happen in an earlier development phase, therefore model-based approaches are necessary.

Model-based SW-FMEA in this sense has been proposed recently [14, 9]. Executable models enable the analysis of software behavior either by simulation, where potential execution traces are sampled and analyzed, or by exhaustive and systematic analysis of the whole state space. The work in [9] is based on the simulation of executable software models with model-level fault injection. An earlier approach, similar to the one presented in this paper, uses model checking (a technique for systematic formal analysis) to detect the violation of the system-level specification in case active faults are present [14]. Similar techniques are used for the analysis of fault tolerance mechanisms. The work described in [6] uses equivalence checking to prove that a fault tolerance mechanism really masks the faults of a fault model, i.e., they check if the fault-free reference model and the faulty system integrated with the mechanism are equivalent. This approach is extended with temporal logic specifications in [7].

The common features of these approaches include a predefined set of faults injected into the code or model, a specification of system-level failures/hazards, and some sort of execution (either testing, simulation, model checking or equivalence checking) to generate traces connecting the first two. The method presented in this paper (in Section 5) refines them by using monitors to define a sort of equivalence relation from the viewpoint of an engineer, as well as by optimizing the model checking process for the efficient handling of nondeterministic fault activations.

2.2 Model Checking

Model checking is an automated formal verification technique used to verify whether a system satisfies a requirement or not. Precisely, given a formal specification φ and a formal behavioral model M of the system, a model checker has to prove or disprove $M \vDash \varphi$ by systematically (and typically exhaustively) analyzing the states and/or possible behaviors of the system model (i.e., the *state space*).

Models are usually given in various high-level formalisms that have to be translated into state-graphs in order to efficiently traverse and analyze them. This process is called state space generation or state space exploration. One of the biggest challenges of model checking arises here: the so-called "state space explosion problem" denotes the usual combinatorial explosion of the size of the state-graph compared to the size of the models of concurrent components, meaning that even compact high level models may produce huge state-graphs, often exponential in size.

There are many families of model checking algorithms described in the literature. An important characterization is based on the class of specification to check. The simplest task is *reachability analysis*, where the algorithm has to decide if a state satisfying certain constraints can be reached from the initial state. *Safety properties* can be reduced to reachability – violating a safety requirement can be demonstrated by a trace leading to an unsafe state. The examples throughout this paper will use assertions to describe simple safety requirements. More complex variants usually check *temporal logic specifications*, the most wide-spread formalisms being linear temporal logic (LTL) and computation-tree logic (CTL).

In this work, the tool SPIN is used as a model checker [15]. SPIN is an explicit-state model checker (using an explicitly stored graph representation of the state space) capable of reachability analysis and linear temporal model checking. Some of its strengths include its maturity, the rich set of configuration opportunities and the expressiveness of its input model, given in PROMELA (PROcess MEta LAnguage). In this paper, the examples will be presented as C code that can easily be transformed into PROMELA due to the similarity of the two languages.

3 Introduction of the Running Example

Throughout the paper, our approach to SW-FMEA will be demonstrated on a running example: a simplified preemptive scheduler (note that on the basis of this example, later it will be straightforward to discuss our industrial case study, the analysis of a real-time operating system). The implementation is given as C code in Listings 1–4. The scheduler has two functions on its interface: **`activate`** and **`stop`**. The third method **`preempt`** is an internal function used by **`activate`** (see below). For the example, assume that the variable current is part of the interface as *read-only* data, but the Boolean arrays **`idle`**, **`waiting`** and **`running`** are for internal use only.

The code is assumed to be called from a single thread of another component of the OS responsible for the coordination and running of the processes. The number of processes is given by the constant **`PROC COUNT`** – in this case, we will use

three processes. Every process is assumed to have a unique ID ranging from 0 to **PROC_COUNT** − 1, which is also interpreted as their priority, the higher value indicating more importance. Every process can be in three states: *Idle*, *Running* and *Waiting*. For the possible state-transitions, see Fig. 1.

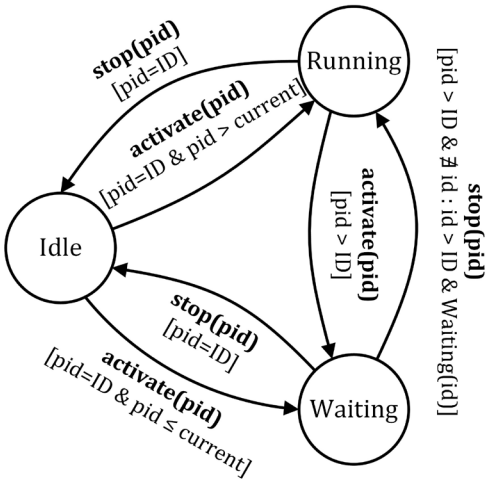


Fig. 1 State machine of a single process.

The global variables encode the state of the scheduler. The variable **current** identifies the running process, or equals −1 if every process is idle. The Boolean arrays encode the states of the processes: the element corresponding to a given process is true if it is in the state corresponding to the array.

When a client calls the function **activate** (Listing 2) with the ID of a process, the scheduler first checks whether there is a running process or not. If not, then the process is started: it is moved to the *Running* state by setting the Boolean variables accordingly and the variable **current** is set to the parameter **pid**. If another process is running, the internal function **preempt** gets called with the same parameter **pid**. The return value indicates the success of activating the process.

The function **preempt** (Listing 3) handles the priorities of processes and ensures that the process with the higher priority gets the right of execution. In case the received parameter **pid** indicates that a process with higher priority wants to run, it moves the current process to the *Waiting* state (note that only the relevant Boolean variables are set), as well as it activates the new process and updates the variable **current**. In case **pid** belongs to a process with lower priority, it is moved into a *Waiting* state and **current** is left unchanged. Again, the return value indicates the success of activating the process belonging to **pid**.

The function **stop** (Listing 4) deactivates the process belonging to **pid** (note that there is no other way a process can return to the *Idle* state). To do this, it updates the internal Boolean arrays accordingly. In addition, if the stopped process was the one running, the function also has to set the variable **current** to −1 and then activate a waiting process, if any. Thus, the function loops through the set of waiting processes in a reverse order with respect to their priorities to find the process with the highest priority and activate it. If no such process is found, the value of **current** remains −1 to indicate that currently there is no running process.

This example is inspired by the OSEK API [1], a common interface definition for safety-critical embedded operating systems (see Section 6 for the description of a pilot project applying the techniques presented in this paper to a PROMELA model of the OSEK API). Throughout the paper, the running example will be referred to as "SimpleScheduler ".

4 Model Checking-based Software FMEA

The approach presented in this paper focuses on the "Effect Analysis" part of FMEA. Assuming a set of possible faults (failure modes on component level) in the software and a characterization of system-level failures, it examines an executable model of the system to generate traces leading to system-level failures.

The process (shown in Fig. 2) starts with fault injection into the system model (often called as model mutation), when the input model is transformed into an analysis model containing faults that can be activated (or deactivated). It is assumed that there is an oracle model that allows the detection of system level failures (see Section 4.2 for details), so the model checker can explore the state space of the analysis model to check if any fault can cause a system-level failure. The output is a set of traces that lead from every relevant fault activation to reachable system-level failures.

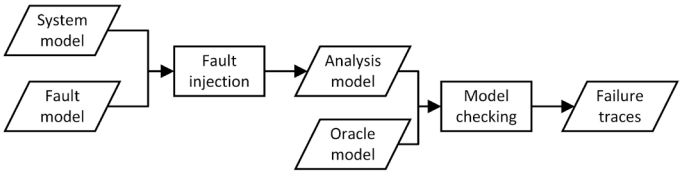


Fig. 2 Overview of the presented approach.

4.1 Fault Injection

The method requires a fault model in terms of the modeling language. Here, a fault model is assumed to be a set of alterations (mutations) that can be activated in the model. The activation is controlled by a *trigger variable*, i.e., with the trigger variable set to *false*, the model should behave correctly, while a value of *true* should activate the corresponding alteration and thus cause an erroneous state when the affected part is executed. Note that trigger variables become part of the system as auxiliary state variables.

Example 1 For the C language there are numerous well-known mutation operators that can inject common coding errors into C programs (e.g., see [23] for a set of traditional syntax-based operators or [13] for the most common software faults). For the sake of simplicity, let us consider four custom operators that resemble some of the most common faults in [13] (also used

Listing 1 Global declarations.

```
const int PROC_COUNT = 3;
//Internal variables
bool [PROC_COUNT] idle = { true, true, true };
bool [PROC_COUNT] waiting = { false, false, false };
bool [PROC_COUNT] running = { false, false, false };
//Interface variables
int current = -1;
```

Listing 2 The `activate` function.

```
bool activate (int pid)
{
  if (current < 0) //System idle
  {
    //State of process: Running
    idle [pid] = false;
    waiting [pid] = false;
    running [pid] = true;
    //Update current process
    current = pid;
    return true;
  }
  else
  {
    //Choose process to run
    return preempt (pid);
  }
}
```

Listing 3 The `preempt` function.

```
bool preempt (int pid)
{
  if (pid > current) //Preempting
  {
    //Running process goes Waiting
    if (current >= 0)
    {
      waiting [current] = true;
      running [current] = false;
    }
    //State of process: Running
    idle[pid] = false;
    waiting[pid] = false;
    running[pid] = true;
    //Update current process
    current = pid;
    return true;
  }
  else
  {
    //State of process: Waiting
    idle[pid] = false;
    waiting[pid] = true;
    running[pid] = false;
    return false;
  }
}
```

Listing 4 The `stop` function.

```
void stop (int pid)
{
  //State of process: Idle
  idle[pid] = true;
  waiting[pid] = false;
  running[pid] = false;
  if ( current == pid ) // Was running
  {
    current = -1;
    //Choose highest priority waiting
    int i = PROC_COUNT - 1;
    while (current < 0 && i >= 0)
    {
      if (waiting[i])
      {
        current = i;
        //New process goes Running
        waiting[current] = false;
        running[current] = true;
      }
      i--;
    }
  }
}
```

in [20]), with the addition of the trigger variable described above. The operators are defined as C preprocessor macros. The macros, as well as simple examples illustrating each operator are presented in Listings 5–8.

Listing 5 shows the simplest fault in our example fault model: if the trigger variable is set, the operand of the *OM operator* is not executed. A similar, but more general fault is presented in Listing 6, where the *ALTSTMNT operator* substitutes the original statement with an altered one. We will use two addi- tional operators in the example: the *NEG operator*, which negates a Boolean expression if activated (Listing 7) and the *ALTEXP operator* (Listing 8), which substitutes an expression with a mutated one of the same type.

Using the trigger variables, a number of different fault types can be modeled. First, a fault can be permanent (only nondeterministic *false* → *true* transitions change the value of the trigger variable) or transient (nondeterministic *true* → *false* transitions are also present). Although in case of software bugs, the faults are usually permanent, it is often useful to have transient faults to simulate the effects of hardware faults as well. Second, it is sometimes desired to restrict the number of faults in the system to at most one, or in some cases at most two, which constrains how the values of trigger variables are allowed to change.

By injecting the fault activation mechanism into the model and having the trigger variables as auxiliary state variables, a model checker is free to choose which fault to activate by setting the trigger variables as long as it meets the restrictions.

Example 2 For example in PROMELA, it is possible to set a variable nondeter- ministically. By injecting a series of such statements into the beginning of the main process definition (i.e., the entry point of the executable model), a model checker is free to choose any valuation of the trigger variables and can explore the behavior of the model accordingly. Restrictions can be applied either by constraining the valuations in the model, or by using an implication in the specification to be checked (*valid valuation* ⇒ *correct behavior*).

4.2 Failure Detection

The traditional approach in model checking is to provide a formal specification of the system to be verified. Automated FMEA can then check if the specification still holds in the presence of faults (as described in [14]).

In this work, we suggest an alternative approach that is closer to a safety engineer 's viewpoint. Instead of specifying a failure (or the correct behavior), it is often easier to include in the model an explicit monitor component to detect and signal requirement violations. Such a monitor can be idealistic (e.g., it may observe every detail of the system or have infinite memory), since its only role is to provide a definition for system-level failures, and it does not have to be implemented in the real system. Due to these properties, we will call this idealistic component an *oracle* (in a similar sense to oracles in testing [26]). Depending on the goals and the domain, an oracle can be a fault-free reference model to compare its behavior with that of the analysis model, a more permissive abstract contract, or even the set of expected outputs for a predefined workload (i.e., a set of test cases). Example oracles will be presented in Section 5.1.

Example 3 The C language provides a simple mechanism for describing invariant properties of a program: assertions. An assertion expresses that at the point of executing an assert statement, its operand *must* be true. Assertions are not meant to be used for error handling, they rather belong to the specification of the problem. In this sense, a set of assertions can be regarded as an oracle.

In our running example, we will use assertions to specify certain safety properties of the SimpleScheduler program. While the code currently does not contain assertions, we will use the workload shown in Listing 9 having an assertion after every function call to check if the implementation behaves as specified (this can again be regarded as an oracle). Note that the assertions only check the variable current and the return value of the functions, as we specified only those as part of the interface and the observable inner state. The workload and the expected behavior (including the values of internal Boolean arrays) is visualized in Fig. 3.

Examples in Section 5 will also use assertions to state when a monitoring component would raise an error. The monitors described in that section can also be regarded as oracles.

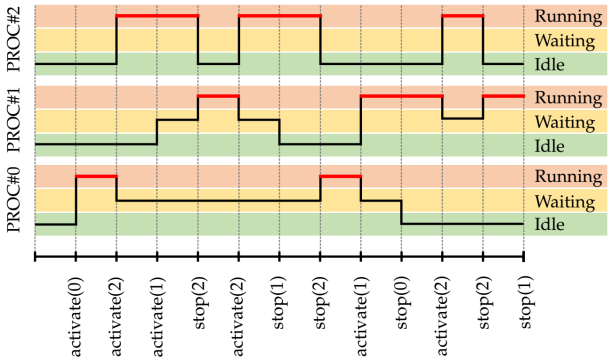


Fig. 3 Timing diagram illustrating the workload and the expected behavior of the scheduler. The value of `current` is denoted with the red lines.

4.3 Failure traces

From the mutated model and the oracle, the model checker will be able to generate a set of traces leading from activated faults to system-level failures. From each trace, we can extract the values of the trigger variables (i.e., which faults were activated) and the location of the system-level failure detected by the oracle. If the oracle can classify the failure (e.g., based on some general failure classification such as in [18] or [20]), this information can also be retrieved.

Listing 5 Omission of a statement.

```
1  //Macro definition for the operator
2  #define OM(stmnt, trigger) \
3  if (!trigger) {stmnt;}
4  //Original code
5  current = pid;
6  //Injected operator
7  bool FAULT_OM = false;
8  OM(current = pid, FAULT_OM);
9  //Expanded operator
10 bool FAULT_OM = false;
11 if (!FAULT_OM)
12   current = pid ;
```

Listing 6 Alternative statement.

```
1  //Macro definition for the operator
2  #define ALTSTMNT (stmnt, alt, trigger) \
3  if (trigger) {stmnt;} \
4  else {alt;}
5  //Original code
6  current = i;
7  //Injected operator
8  bool FAULT_ALTSTMNT = false;
9  ALTSTMNT (current = i, current = pid, FAULT_ALTSTMNT);
10 //Expanded operator
11 bool FAULT_ALTSTMNT = false;
12 if (FAULT_ALTSTMNT) { current = pid; }
13   else { current = i; }
```

Listing 7 Negation of a Boolean expression.

```
1  //Macro definition for the operator
2  #define NEG(exp, trigger) \
3  (trigger ? !(exp) : (exp))
4  //Original code
5  waiting[current] = true;
6  //Injected operator
7  bool FAULT_NEG = false;
8  waiting[current] = NEG(true, FAULT_NEG);
9  //Expanded operator
10 bool FAULT_NEG = false;
11 waiting[current] =
12   (FAULT_NEG ? !(true) : (true));
```

Listing 8 Alternative expression.

```
1  //Macro definition for the operator
2  #define ALTEXP(exp, alt, trigger) (trigger ? (alt) : (exp))
3  //Original code
4  if (pid > current)
5  { ... }
6  //Injected operator
7  bool FAULT_ALTEXP = false;
8  if (ALTEXP (pid > current, pid > current, FAULT_ALTEXP)
9  { ... }
10 //Expanded operator
11 bool FAULT_ALTEXP = false;
12 if (FAULT_ALTEXP ? pid < current : pid > current)
13 { ... }
```

Listing 9 The workload and the expected behavior.

```
1  int main()
2  {
3    bool ret;
4    ret = activate(0);   assert(current == 0 && ret == true);
5    ret = activate(2);   assert(current == 2 && ret == true);
6    ret = activate(1);   assert(current == 2 && ret == false);
7    stop(2);             assert(current == 1);
8    ret = activate(2);   assert(current == 2 && ret == true);
9    stop(1);             assert(current == 2);
10   stop(2);             assert(current == 0);
11   ret = activate(1);   assert(current == 1 && ret == true);
12   stop(0);             assert(current == 1);
13   ret = activate(2);   assert(current == 2 && ret == true);
14   stop(2);             assert(current == 1);
15   stop(1);             assert(current == -1);
16 }
```

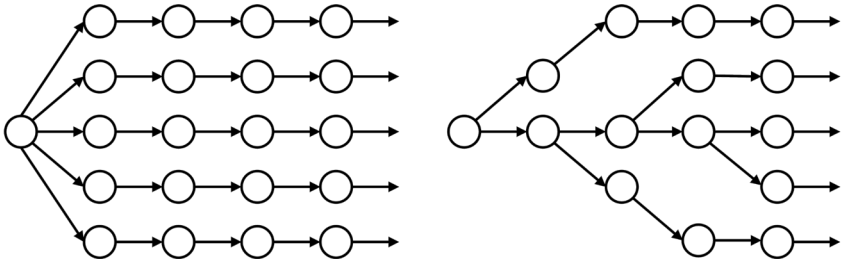


Fig. 4 Shape of the state space with eager and lazy evaluation.

Example 4 Assume that the operating system using the SimpleScheduler will observe the variable current to determine which process to run, while the return value of the functions is only a secondary information used for administrative purposes (e.g., logging). In this setting, the incorrect value of the variable **`current`** is a severe system-level failure, while a wrong return value is more moderate and has less severe consequences.

Considering a *NEG*-mutation of the return statement in **`activate`** (line 11 in Listing 2), running the workload (Listing 9) will produce an assertion failure immediately after the first **`activate`** call, providing a trace leading from a fault-activation (setting the corresponding trigger variable to **`true`**) to the system-level failure (the failed assertion). From the trace, it is easy to obtain the activated fault, and also the failed assertion, which will classify the failure as moderate.

The described process can be repeated for every fault, which takes us back to simulation-based SW-FMEA [9]. With a model checker, however, it is possible to analyze the effects of every fault together, which allows us to greatly speed up the process.

4.4 Efficiency and Lazy Evaluation

This section presents a method that can advance model checking-based SW-FMEA beyond a set of automated simulations. The presented method can be implemented either by altering the model to achieve the desired effect, or by adapting the model checker to apply the method automatically.

From the technical point of view, the problem is that model checking is highly sensitive to the size and potential values of the state vector. Unfortunately, adding a set of nondeterministic boolean variables (here the trigger variables) increases the number of potential states exponentially. Moreover, if permanent faults are modeled in such a way that the initial activation is random, the number of initial states immediately blows up exponentially.

In order to avoid the combinatoric explosion, in our method we suggest a "lazy" strategy to evaluate fault activations. Let the trigger variables have ternary values, with the third value being *undefined*, also being the initial value. By injecting additional logic to access and evaluate the value of trigger variables, it is possible to defer the valuation and have identical states for multiple fault configurations up to an actual fault activation, from which point the error will propagate separately.

This effect is illustrated in Fig. 4. Assume that every branch in the state space is the result of deciding fault activations. When trigger variables are evaluated in an "eager " way, there is only a single decision at the beginning of the program, setting at most one trigger to *true*. The rest of the execution is deterministic, therefore the shape of the state space is similar to the left figure.

In case of lazy evaluation, the activation of a fault is not decided until the corresponding code is executed. This means that when the first alteration is hit, a binary decision will occur: should the fault become activated or not? If it is activated, the program is again deterministic under the single fault assumption. Otherwise, the program is run to the next decision point,

where the process repeats. The shape on the right of Fig. 4 is the result of not distinguishing between an activated and an inactive fault until the point this distinction becomes important, i.e., then the affected code is executed and an error is caused.

Example 5 To implement the lazy evaluation method by altering the model, the macros defined in Listings 5–8 can be extended. Since C does not include any constructs to describe nondeterministic choice and value assignment (for obvious reasons), let us assume that the function `nondet_val(...)` returns one of its arguments in an unpredictable (nondeterministic) way. From a programmer point of view, such a function can be regarded as external input, but in formal modeling languages, such as PROMELA, nondeterministic behavior can be expressed and analyzed. Using this artificial function and an `enum` type instead of the Boolean trigger variables, the extended macros are shown in Listing 10. The presented version enforces that at most one fault can be activated in a permanent (non-transient) way. The evaluation mechanism is given both as C macro and as a C++ function for the sake of clarity (both of them can be translated to PROMELA).

4.5 Example: Analyzing the SimpleScheduler

The previous sections presented the general idea of our approach, along with detailed examples describing the specific application of the process on the SimpleScheduler example. To demonstrate the benefits of our automated SW-FMEA framework, this section combines the previous examples to complete the analysis of the SimpleScheduler.

So far, we have seen four mutation operators defined as C-style macros (also available in PROMELA), implementing the lazy evaluation strategy and the constraint of having at most one (permanent) fault activation. To evaluate the fault effects in case of the SimpleScheduler (which is not yet prepared to handle any faults), we need to instantiate the mutation operators for the code shown in Listings 1–4.

Listing 10 Lazy evaluation of triggers.

```
1  //Ternary trigger type, the default value should be Undefined
2  enum t_trigger { Undefined, True, False }
3  //Any fault activated?
4  bool faulty = false;
5
6  //Lazy evaluation of the ternary trigger
7  #define EVAL(trigger) \
8  ((trigger == Undefined ? \
9    trigger = (faulty ? False : nondet_val (True, False)) : \
10   trigger), \
11 (faulty = faulty || trigger), \
12 (trigger == True ? true : false))
13
14 //Implemented as a function (with C ++ reference type)
15 bool EVAL(t_trigger & trigger)
16 {
17   if (trigger == Undefined)
18     if (faulty)
19       trigger = False;
20     else
21       trigger = nondet_val(True, False);
22   faulty = faulty || trigger;
23   if (trigger == True)
24     return true;
25   else
26     return false;
27 }
28
29 //The omission operator
30 #define OM(stmnt, trigger) \
31 if (!(EVAL(trigger)) {stmnt;}
32
33 //The alternative statement operator
34 #define ALTSTMNT(stmnt, alt, trigger) \
35 if (EVAL(trigger)) {stmnt;} \
36 else {alt;}
37
38 //The negation operator
39 # define NEG(exp, trigger) \
40 ((EVAL(trigger)) ? !(exp) : (exp))
41
42 //The alternative expression operator
43 #define ALTEXP(exp, alt, trigger) ((EVAL(trigger)) ? (alt) : (exp))
```

Table 1 Results of SW-FMEA on the SimpleScheduler. The marked faults can cause a system-level failure.

ID of activated fault	Failure	ID of activated fault	Failure
ACTIVATE_NEG_3	*X*	PREEMPT_OM_9	
ACTIVATE_ALTSTMNT_16	*X*	PREEMPT_OM_12	
ACTIVATE_NEG_6		PREEMPT_OM_13	
ACTIVATE_NEG_7		PREEMPT_OM_14	
ACTIVATE_NEG_8		PREEMPT_OM_16	*X*
ACTIVATE_NEG_11	*X*	PREEMPT_OM_22	
ACTIVATE_OM_6		PREEMPT_OM_23	*X*
ACTIVATE_OM_7		PREEMPT_OM_24	
ACTIVATE_OM_8		STOP_NEG_7	*X*
ACTIVATE_OM_10	*X*	STOP_ALTSTMNT_11_22	*X*
ACTIVATE_OM_16	*X*	STOP_NEG_4	
PREEMPT_ALTEXP_3	*X*	STOP_NEG_5	*X*
PREEMPT_NEG_8	*X*	STOP_NEG_6	
PREEMPT_NEG_9		STOP_NEG_14	*X*
PREEMPT_NEG_12		STOP_NEG_18	
PREEMPT_NEG_13		STOP_NEG_19	
PREEMPT_NEG_14		STOP_OM_4	
PREEMPT_NEG_17	*X*	STOP_OM_5	*X*
PREEMPT_NEG_22		STOP_OM_6	
PREEMPT_NEG_23	*X*	STOP_OM_9	*X*
PREEMPT_NEG_24		STOP_OM_16	*X*
PREEMPT_NEG_25	*X*	STOP_OM_18	
PREEMPT_OM_8	*X*	STOP_OM_19	

For the example, we chose to automatically instantiate an *OM* mutation for every simple instruction and a *NEG* mutation for every Boolean expression. The only exception is the condition on line 3 of **`preempt`**, where instead of a *NEG* mutation, we applied the *ALTEXP* operator to mimic the semantic fault of using the wrong relation operator (e.g., '<' instead of '>'). We also used two instances of the *ALTSTMNT* operator: once on line 16 of the **`activate`** function to call the **`preempt`** function with the variable **`current`** instead of the parameter **`pid`**, and in the other case to use a forward loop instead of the correct backward loop on lines 11–22 of the **`stop`** function. Every alteration was assigned an identifier in the form *<function name>_<mutation operator>_<affected line(s)>* (for example, a *NEG* operator affecting the expression on line 3 of the **`activate`** function has the identifier ACTIVATE_NEG_3.

We will use the workload defined in Listing 9 together with the specified oracle (the assertions), this time not differentiating between the various assertion violations. After running SPIN on the generated PROMELA model with the injected mutations, the tool returned a set of traces leading to assertion violations. As the only nondeterminism in the model is introduced by the permanent fault activations, there could be a single trace for each fault that causes an assertion failure. For each trace, the last values of trigger variables can tell which fault activation the trace belongs to – this information is extracted into Table 1.

It is interesting to discuss the faults that did not cause an assertion failure. Only *NEG* and *OM* operators introduced non-critical faults (i.e., faults that did not cause a failure), all of which affected lines setting the internal variables encoding the states of the processes. Only the variable encoding the *Waiting* state seems to be critical, as its wrong or missing assignment still causes some failures (and setting it to true seems to be more important). This should not be surprising, since only the value of this variable is read and used in the functions, specifically in the function stop.

Even in this small example, it is worth to analyze and interpret the results. Constructs like the encoding of process states – that look natural at first – may turn out to be redundant or surprisingly critical. This is even more so if the system is large (thus the effects of a design decision are not easy to comprehend), so the techniques used here also get more beneficial as complexity grows.

5 Evaluation of Fault Tolerance Mechanisms and Error Detectors

Section 4 outlined a general approach to model checking-based automated SW-FMEA. In this section, we present two applications of the method to evaluate fault tolerance

mechanisms and error detectors. Once the effects of a fault are discovered and understood, the next step (usually also part of the FMEA) is to design a mechanism for the detection and the mitigation of errors caused by the activation of the fault [18]. Using such mechanisms, it is possible to wrap "risky" components to raise the overall reliability of the system [3]. Naturally, it is therefore equally important to evaluate the chosen techniques. The goal of this section is to provide a measure for the efficiency of error detection and fault-tolerance mechanisms by analyzing how many and what type of faults they can detect or mask, respectively.

For an "absolute" measure, one can use an idealistic oracle (like the oracle we suggested in Section 4.2) as a baseline and "upper bound" on the efficiency of realistic approaches. In case of error detectors, it is also possible to compute the *relative efficiency* of two solutions, i.e., how much "better " or "worse" is one of them compared to the other.

The measurement setting is the following. In case of error detectors, we first run a check with the oracle to get the total number of traces leading to observable failures (denoted T as *total*), then we measure the same number (denoted D as *detected*) with the evaluated detector (Fig. 5). Relative efficiency is obtained by using another error detector as reference (to compute T) instead of the oracle. In case of fault tolerance mechanisms, both steps use the oracle, with the mechanism integrated with the system in the second step (Fig. 6). Efficiency is then defined as follows.

- In case of error detectors, the efficiency is $E = D/T$.
- In case of fault tolerance mechanisms, the efficiency is $E = (T - D)/T$.

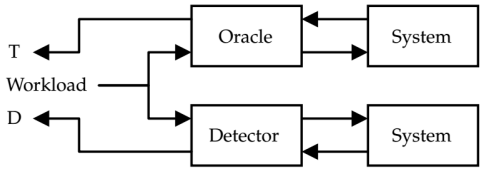


Fig. 5 Measurement setting for error detectors.

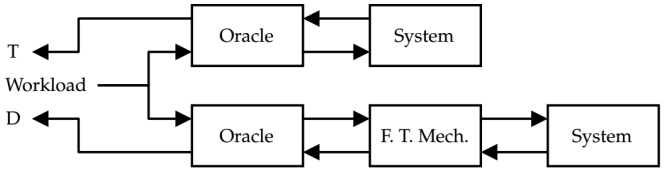


Fig. 6 Measurement setting for fault tolerance mechanisms.

Efficiency can also be defined for each fault type (or failure mode), giving a more detailed picture about the evaluated technique. By obtaining a number describing the efficiency of different approaches, we hope to help design decisions concerning what error detectors and fault tolerance mechanisms to use (possibly in some combination).

The following sections present detailed examples of applying and computing the introduced measures with the motivation of giving the reader a clear picture of the details and the benefits of using our approach. Readers not interested in the details of application are advised to skip the rest of Section 5 and continue with Section 6 where our industrial case study is described.

5.1 Example: Error Detectors

To illustrate the use of model checking-based SW-FMEA for the evaluation of error detectors, we have designed monitors for the SimpleScheduler example. This section shortly summarizes the implemented monitors to give some ideas about potential error detection techniques, then the results of the comparison – the efficiencies – are presented with detailed explanation.

5.1.1 The Master-Checker Monitor

The Master-Checker architecture uses a replica of the system to perform the same computation. The replica is often called a "Checker", because its output is used to check the validity of the "Master" implementation – if they do not agree, the system signals and error. In this paper, we will refer to the "Checker" as the fault-free Reference Model (abbreviated as *REF*).

The Reference Model will be the same code (the implementation of the SimpleScheduler), but without any mutations injected. The Master-Checker monitor code first runs the Master code, then the Checker, after which it compares the return values and the values of interface and internal variables. If any deviation is detected, the monitor signals an error, i.e., an assertion failure will occur.

Note that in this form, the Reference Model is an appropriate choice to measure the upper bound of detectable faults.

5.1.2 The State Machine Monitor

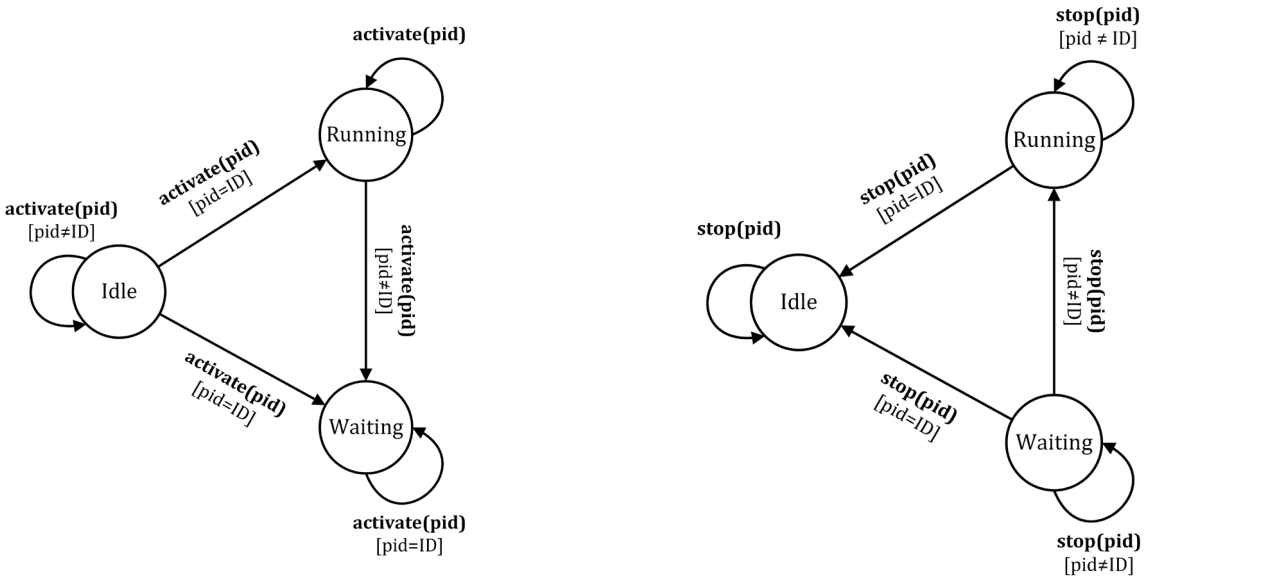
The State Machine monitor (abbreviated as *SM*) checks conformance with a more abstract version of the state machine shown in Fig. 1. Information about priorities and other processes are not taken into account, resulting in the state machine presented in Fig. 7 (for the sake of simplicity, transitions triggered by the two functions `activate` and `stop` are depicted separately).

The State Machine monitor maintains internal Boolean variables to store the last state of the processes. After each call and for every process, it checks *1)* if the process is in a valid state (i.e., only one of the corresponding Boolean variables is *true*), and *2)* if there is an allowed transition between the last and the current state.

5.1.3 The Interface Contract Monitor

The Interface Contract monitor (abbreviated as *IC*) checks a set of requirements after each function call:

- After `activate`, if the process to be activated has higher priority than the process running before the call, it must be the one currently running.



(a) Allowed for **`activate`** calls

(b) Allowed for **`stop`** calls.

Fig. 7 Transitions allowed by the *SM* monitor.

- After **`activate`**, the return value is true *iff* the activated process has higher priority than the process running before the call.
- After **`stop`**, the new running process (if any) cannot have a higher priority than the one previously running (otherwise there would have been a priority inversion, since the currently running process would have been waiting for another process with lower priority).

The monitor requires a single integer variable storing the ID of last running process, which is updated after each call.

5.1.4 The Priority Checker Monitor

The Priority Checker monitor (abbreviated as *PC*) performs the same check after each function call: there should be no process that is in the *Waiting* state and has a higher priority than the currently running process.

The main difference between the Interface Contract and the Priority Checker is that the Interface Contract uses only the information available on the interface (i.e., the return value and the public variable current), while the Priority Checker observes the global state of the scheduler to monitor priority inversions.

5.1.5 Results

The results of running the model checker with the measurement setting presented in Fig. 5 and analyzing the generated traces is shown in Table 2.

The first column shows the ID of the fault as introduced in Section 4.5. The second column belongs to the Master-Checker oracle (or Reference Model), which also serves as the baseline of comparison (*T*), while the remaining columns show results for the other three detectors and their combination (abbreviated as *CMB*). A cross in a cell denotes that the corresponding detector generated an error signal when the corresponding fault was activated. The two bottom rows summarize the performance of detectors with the number of faults detected and the efficiency computed with *REF* as the baseline.

The table has a number of interesting implications:

- The Reference Model (*REF*) failed to detect the activation of three faults: the omission of line 7 in **`activate`** and lines 13 and 24 in **`preempt`**. Both of these instructions set an internal Boolean variable that corresponds to a state that is neither left nor entered in that particular call, so it is not surprising that their omission does not cause a system-level failure – indeed it is safe to say that they are redundant.
- The State Machine monitor (*SM*) detected most of the faults related to the internal Boolean variables associated with the states of the processes.
- The Interface Contract (*IC*) detected most of the faults affecting the han- dling of variable current, but had no success with the stop function. This is due to the fact that the interface does not provide enough information to constrain the next running process.
- The Priority Checker (*PC*) detected problems related to the usage of variable current, the setting of the Boolean variable corresponding to the *Waiting* state, and the selection of the next process after stopping one.
- The combination of simpler detectors (*CMB*) is almost as efficient as the idealistic Reference Model. It fails to detect only the omission of line 16 in stop, which sets the variable current to the selected new process. Note that the combination of error detectors detects an error *iff* one of the constituent detectors can detect it.

Table 2 Efficiency of error detectors. More detected errors is better.

ID	REF	SM	IC	PC	CMB
ACTIVATE_NEG_3	*X*		*X*		*X*
ACTIVATE_ALTSTMNT_16	*X*	*X*	*X*	*X*	*X*
ACTIVATE_NEG_6	*X*	*X*			*X*
ACTIVATE_NEG_7	*X*	*X*		*X*	*X*
ACTIVATE_NEG_8	*X*	*X*			*X*
ACTIVATE_NEG_11	*X*		*X*		*X*
ACTIVATE_OM_6	*X*	*X*			*X*
ACTIVATE_OM_7	–	–	–	–	–
ACTIVATE_OM_8	*X*	*X*			*X*
ACTIVATE_OM_10	*X*		*X*		*X*
ACTIVATE_OM_16	*X*	*X*	*X*		*X*
PREEMPT_ALTEXP_3	*X*		*X*	*X*	*X*
PREEMPT_NEG_8	*X*	*X*			*X*
PREEMPT_NEG_9	*X*	*X*			*X*
PREEMPT_NEG_12	*X*	*X*			*X*
PREEMPT_NEG_13	*X*	*X*		*X*	*X*
PREEMPT_NEG_14	*X*	*X*			*X*
PREEMPT_NEG_17	*X*	*X*	*X*		*X*
PREEMPT_NEG_22	*X*	*X*			*X*
PREEMPT_NEG_23	*X*	*X*			*X*
PREEMPT_NEG_24	*X*	*X*			*X*
PREEMPT_NEG_25	*X*		*X*		*X*
PREEMPT_OM_8	*X*	*X*			*X*
PREEMPT_OM_9	*X*	*X*			*X*
PREEMPT_OM_12	*X*	*X*			*X*
PREEMPT_OM_13	–	–	–	–	–
PREEMPT_OM_14	*X*	*X*			*X*
PREEMPT_OM_16	*X*		*X*	*X*	*X*
PREEMPT_OM_22	*X*	*X*			*X*
PREEMPT_OM_23	*X*	*X*			*X*
PREEMPT_OM_24	–	–	–	–	–
STOP_NEG_7	*X*			*X*	*X*
STOP_ALTSTMNT_11_22	*X*			*X*	*X*
STOP_NEG_4	*X*	*X*			*X*
STOP_NEG_5	*X*	*X*		*X*	*X*
STOP_NEG_6	*X*	*X*			*X*
STOP_NEG_14	*X*	*X*		*X*	*X*
STOP_NEG_18	*X*	*X*		*X*	*X*
STOP_NEG_19	*X*	*X*			*X*
STOP_OM_4	*X*	*X*			*X*
STOP_OM_5	*X*	*X*		*X*	*X*
STOP_OM_6	*X*	*X*			*X*
STOP_OM_9	*X*			*X*	*X*
STOP_OM_16	*X*				
STOP_OM_18	*X*	*X*		*X*	*X*
STOP_OM_19	*X*	*X*			*X*
Total detected [D]:	43[=T]	33	9	13	42
Efficiency [D/T]:	100%	76.7%	20.9%	30.2%	97.7%

It is beneficial to investigate the missed fault of the combined detector a bit further. The problem is that the *SM* monitor does not see the value of **`current`**, the *IC* does not have enough information to constrain the new value (which will be -1 if line 16 is omitted) and since **`current`** is never changed, the loop will set every waiting process to the *Running* state which will cause the *PC* to detect no waiting processes at all.

In this case, it is now obvious that the *SM* monitor (or a new one) should also check if at most one process is running, but this was not clear before running the analysis. Indeed, one of the main benefits of applying our method is the better understanding of the system and the monitors, facilitating an iterative process that constantly raises the quality of the system.

5.2 Example: Fault Tolerance Mechanisms

We have also designed fault tolerance mechanisms for the SimpleScheduler example to illustrate the notion of efficiency in this case. These mechanisms are implemented as components whose job is to mask the effects of faults before they propagate to other components [3]. To do this, they can alter the output and internal state of the system to maintain the most critical properties that must be satisfied.

The work published in [8] call this type of component a *shield*, because it protects the system from the most critical failures at the cost of sacrificing less important properties and functions. There are other, more heavyweight approaches, such as N-version programming [11], which does not sacrifice any behavior, but reduces failures by the means of voting. The following examples are similar to shields, but in a less formal way than described in [8].

5.2.1 Fixing Priority Problems

This component (abbreviated as *FIXP*) attempts to repair priority inversions by modifying the variable **`current`** if any problem is detected. Specifically:

- if after activation of a process, the currently running one has a lower priority than the one to have been activated, it modifies the variable current and the return value to reflect that the newly activated process should be running;
- if after activation, the currently running process has a lower priority than the previous one (if any), the previous process is restored as running and the return value is set to **`false`** to reflect that the new process should not have been activated;
- if after stopping a process, it is still set as currently running, the waiting process with highest priority is selected and the **`variable`** current is set to its ID.

Note that this shield is only concerned about the publicly accessible interface and does not alter the internal variables. The idea behind such a component is that its code is simple, reducing the chance of implementation errors, so it is usually safe to assume it functions correctly.

5.2.2 Synchronizing Variables and States

Another method to mask faults can be the synchronization of variables and states (abbreviated as *SYNC*). The variable **`current`** and the internal Boolean variables are not independent, so it is sometimes possible restore the states of the processes. The component has two goals:

- move the process identified by the variable **`current`** to the *Running* state by setting the corresponding Boolean variable (and unset it for every other process);
- make sure the encoding is valid for every process by unsetting variables that have not become *true* after the **`activate`** call (prioritizing the *Running* state in case of a conflict).

To do this, the component maintains a copy of the internal Boolean variables to store the last state of the processes (this enables the detection of what changed during the function call). Note that this strategy relies on the assumption of having at most one fault, as this way, it is not possible to achieve a different, but valid state for a process (it requires two instructions to unset the Boolean variable encoding the old state and then to set the new one).

5.2.3 Combined Strategy

We have also evaluated the combination of the *FIXP* and *SYNC* strategies (abbreviated as *COMB*), because they address different parts of the system. The combined component first runs the *FIXP* strategy to restore the currently running process based on priorities, then it applies the *SYNC* strategy to adjust the internal Boolean variables accordingly.

5.2.4 Results

The results of running the model checker with the measurement setting presented in Fig. 6 and analyzing the generated traces is shown in Table 3.

The first column again shows the ID of the fault as introduced in Section 4.5. The second column still belongs to the Reference Model oracle, which will again serve as the baseline of comparison (*T*). The remaining columns show the faults that the Reference Model could not detect anymore when the system was integrated with a certain fault tolerance mechanism. A check mark in a cell denotes that the Reference Model did not generate an error signal (assertion violation) when the corresponding fault was activated, meaning that the given mechanism did its job – the goal is to mask the effect of as many faults as possible. The two bottom rows summarizes the performance of detectors with the number of faults masked (compared to the Reference Model) and the corresponding efficiency.

The implications of Table 3 are as follows:

- The *FIXP* strategy does not seem to be efficient on its own. It can mask only two faults: both of them are the omission of the setting of the variable **`current`** to the

Table 3 Efficiency of fault tolerance mechanisms. More mitigated faults is better.

ID	REF	FIXP	SYNC	COMB
ACTIVATE_NEG_3	*X*			
ACTIVATE_ALTSTMNT_16	*X*			
ACTIVATE_NEG_6	*X*		✓	✓
ACTIVATE_NEG_7	*X*			
ACTIVATE_NEG_8	*X*		✓	✓
ACTIVATE_NEG_11	*X*			
ACTIVATE_OM_6	*X*		✓	✓
ACTIVATE_OM_7	–	–	–	–
ACTIVATE_OM_8	*X*		✓	✓
ACTIVATE_OM_10	*X*	✓		✓
ACTIVATE_OM_16	*X*			
PREEMPT_ALTEXP_3	*X*			
PREEMPT_NEG_8	*X*			
PREEMPT_NEG_9	*X*		✓	✓
PREEMPT_NEG_12	*X*		✓	✓
PREEMPT_NEG_13	*X*			
PREEMPT_NEG_14	*X*		✓	✓
PREEMPT_NEG_17	*X*			
PREEMPT_NEG_22	*X*		✓	✓
PREEMPT_NEG_23	*X*			
PREEMPT_NEG_24	*X*		✓	✓
PREEMPT_NEG_25	*X*			
PREEMPT_OM_8	*X*			
PREEMPT_OM_9	*X*		✓	✓
PREEMPT_OM_12	*X*		✓	✓
PREEMPT_OM_13	–	–	–	–
PREEMPT_OM_14	*X*		✓	✓
PREEMPT_OM_16	*X*	✓		✓
PREEMPT_OM_22	*X*		✓	✓
PREEMPT_OM_23	*X*			
PREEMPT_OM_24	–	–	–	–
STOP_NEG_7	*X*			
STOP_ALTSTMNT_11_22	*X*			
STOP_NEG_4	*X*			
STOP_NEG_5	*X*			✓
STOP_NEG_6	*X*		✓	✓
STOP_NEG_14	*X*			
STOP_NEG_18	*X*		✓	✓
STOP_NEG_19	*X*		✓	✓
STOP_OM_4	*X*			
STOP_OM_5	*X*		✓	✓
STOP_OM_6	*X*		✓	✓
STOP_OM_9	*X*			✓
STOP_OM_16	*X*			
STOP_OM_18	*X*		✓	✓
STOP_OM_19	*X*		✓	✓
Total mitigated [T − D]:	0	2	20	24
Efficiency [(T − D)/T]:	0%	4.7%	46.5%	55.8%

ID of the new process (lines 10 and 16 in **`activate`** and **`preempt`**, respectively). Note though, that these faults would cause immediate and serious failures.

- The *SYNC* strategy masks most of the faults related to the management of internal Boolean state variables, except those that correspond to setting a variable to **`false`** instead of **`true`**. This is because the component relies on the detection of changing a value of **`false`** to **`true`**. Nevertheless, 46.5% of the faults were masked, which is almost ten times better than the efficiency of the *FIXP* strategy.
- The combined strategy (*COMB*) masks everything the two constituent strategies can mask, and even more. Note the masking of STOP_NEG_5 (line 5 in stop) and STOP_OM_9 (line 9 in stop), which were not masked by either strategy, but the combination is now able to negate their effects.

Again, it is beneficial to investigate these last two masked faults, this time by analyzing the sequence of actions after triggering the fault. In case of STOP_NEG_5, the following will happen.

1. The fault STOP_NEG_5 is the wrong assignment of the Boolean variable corresponding to the *Waiting* state, which will cause the stopped process to be idle and waiting *at the same time*.
2. This will cause to loop to find process just stopped as the waiting process with the highest priority, setting it as the currently running process and moving it from the *Waiting* state to the *Running* state (not changing the active *Idle* state).
3. The *FIXP* strategy notices that the same process is still running after stopping it, so it performs the selection itself (this time correctly).
4. At this point, the Reference Model would still signal an error, because the current process is in the *Idle* and *Running* state at the same time, and the Boolean state variables of the selected process is not adjusted by the *FIXP* strategy.
5. However, the *SYNC* strategy will now fix the state encoding of each process, removing the stopped process from the *Running* state and leaving the *Idle* state active (because it indeed changed its value).

The effect of STOP_OM_9 is very similar.

1. The activation of STOP_OM_9 will cause the variable current to not reset to -1. For this reason, the loop will never start, leaving the stopped process as the one currently running.
2. The *FIXP* strategy again notices that the same process is still running after stopping it, so it performs the selection itself.
3. At this point, the Reference Model would again signal an error, because the Boolean state variables of the selected process is not adjusted by the *FIXP* strategy.
4. By applying the *SYNC* strategy as well, the state encoding of each process is again fixed, adjusting the state of the newly activated process.

This example again showed that, in addition of assessing the efficiency of different methods, the analysis of fault tolerance mechanisms together with the system helps in understanding the behavior of the system itself and the strategies applied to mask errors.

6 OSEK API – An Industrial Case Study

In addition to the detailed running example, we also started to conduct a more realistic case study. To demonstrate the merits of the proposed approach to industrial partners, we used the model of the OSEK API [1], a common interface definition for safety-critical embedded operating systems. In a related project[1], an OSEK-compliant real-time operating system targeting the automotive industry was investigated to add fault tolerance and monitoring techniques addressing potential programming faults, both from the side of the OS and client applications. To aid design decisions, we have employed the presented approach to evaluate different solutions still in the modeling phase. For the analysis, we used a model of the OSEK API and a set of test programs (both correct and incorrect) as workload taken from [27] and a set of error detectors with assertions providing the "error signals".

The OSEK API provides a set of interface functions with their syntaxes and also the semantics of the implemented OS primitives. The API defines primitives for task handling and scheduling; resource, interrupt and event handling; semaphores and messaging; as well as timers and alarms. For the case study, we used a model describing the Task API, the Resource API and the Event API. The SimpleScheduler running example can be regarded as a simplified version of this model, without events, resources, and a simplified internal behavior.

We have modeled two types of error detectors: as the idealistic baseline oracle, a (fault-free) Reference Model that was compared to the state of the OSEK model after each interface call (see Section 5.1.1 for the main idea); as well as a Priority Checker that observes only the priorities of scheduled tasks

1 This work has been partially supported by the CECRIS project, FP7–Marie Curie (IAPP) number 324334.

(similar to the combination of the *IC* and *PC* monitors presented in Sections 5.1.3 and 5.1.4). The Priority Checker could detect if the scheduler violated the priorities, for example by preempting a task to run another one with lower priority.

6.1 Implementation of Automated SW-FMEA

Like in the case of the SimpleScheduler example, we have implemented the ap- proach described in Section 4 based on the model checker SPIN. Fault injection was performed by an auxiliary program that parsed the PROMELA model of the OSEK API and altered the code. For this first stage, we used a simple fault model: only *OM*-type mutations were injected into the model (see Section 4.1). We assumed a single, permanent fault that is activated in the initial state (with lazy evaluation built into the model as before).

SPIN was configured to perform a bounded depth-first search optimized for safety checking and enumerating every violating trace. The tool looked for assertion violations (errors detected by the evaluated error detectors) and also invalid end states (i.e. deadlocks). Once the model checking finished, the path was replayed to obtain the last (violating) state, containing the values of the trigger variables and the location of the error signal. This information was aggregated for all traces, resulting in the number of violating traces for each different fault-type.

We have measured the net time of model checking during the analysis process to evaluate the effects of lazy evaluation. Without lazy evaluation, the model checker ran for a total of 590.52 seconds, while the optimized model took 257.81 seconds to check, 56.34% less than before. This gain varied between 12.84% (from 5.2s to 4.5s) and 76.27% (from 85s to 20s) depending on the workload, with a general tendency that workloads producing a higher run time benefit more from lazy evaluation (see Fig. 8 for a comparison of lazy and eager evaluation execution times for every test case). This is in line with our expectations, that is, with more trigger variables and more complex workloads (which is unavoidable with real-life projects), the speedup is greater.

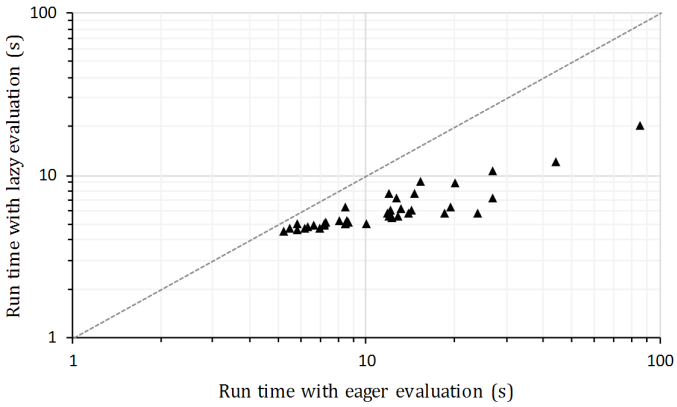


Fig. 8 Comparison of lazy and eager evaluation in terms of execution time with logarithmic scales. The dashed line denotes the diagonal.

6.2 Results

Running the analysis with the two detectors showed the relative efficiency of the Priority Checker compared to the more "heavyweight" and rather idealistic Reference Model. The diagram in Fig. 9 illustrates the efficiency for each fault type (alteration in the API model) separately, also grouping them based on the related API. In this study, multiple test cases were used, so it is possible to compute the efficiency for every mutation instance.

Although the fault model is simple, the diagram highlights that the Priority Checker can barely detect faults in the resource handling or task termination primitives, but it is comparable to the Reference Model for most of the faults related to rescheduling (starting tasks and handling events).

In a real world example, analysis of the characteristics of different detectors could help in understanding their efficiency (or coverage) better. Both in this study and the running example, the monitoring solutions have different complexity and cost. By knowing the costs of a solution and its characteristics (revealed by the proposed analysis approach), it should be easier for engineers to find a cost-optimal solution with the highest possible benefits.

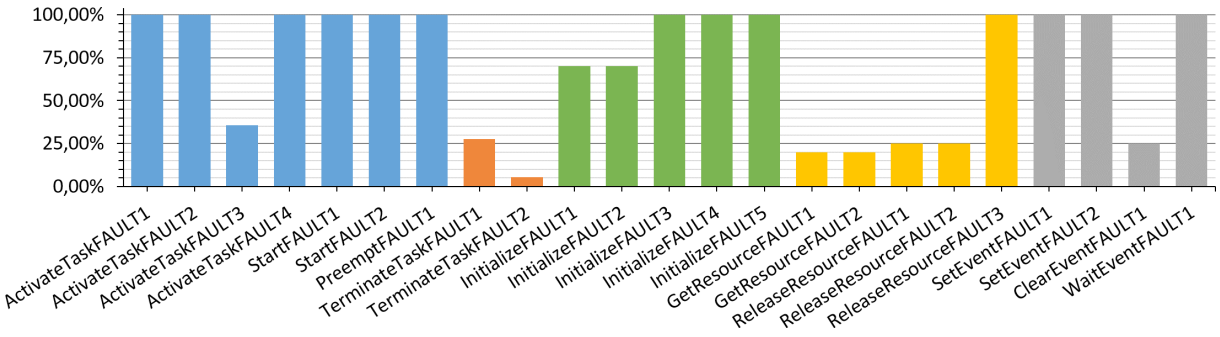


Fig. 9 Efficiency of the Priority Checker compared to the Reference Model

7 Conclusion and Future Work

This paper presented a method for automated SW-FMEA based on model checking, applying the approach in the evaluation of fault-tolerance mechanisms and error detectors. All of the concepts and techniques were demonstrated on a running example derived from the OSEK API definition, proving the benefits and the feasibility of the proposed approach, as well as providing enough details about the main ideas to help in their adaptation.

The main idea of the model checking-based method is to *1)* use model-level fault injection (or model mutations) with trigger variables to augment the system model with faults that can be activated, then *2)* use formal specification or an oracle model to characterize system-level failures so that *3)* the model checker can generate traces leading from a fault activation to a failure. This approach is improved by lazy evaluation, which is a technique to reduce the size of the state space.

Evaluation of fault-tolerance mechanisms and error detectors is based on the notion of (relative) efficiency that describes the number of masked/revealed errors compared to an oracle or another technique (respectively). To the best of our knowledge, there is no other work addressing the evaluation of such techniques in a model-based FMEA setting. We have showed how this additional information can aid safety-engineers in early design decisions.

The main contribution of this paper is the outline of a general idea. In order to adapt it to a certain problem, there are a number of concerns to be considered case by case, including the modeling language, the mutation operators used, and the type of model checker to employ. We hope to help this process by having provided the most important concepts and points of interests through a detailed running example.

We have identified several topics for future work. First, the fault model for executable software models has a great impact on the validity of the re- sults, so a fine-tuned and validated fault model is necessary. We plan to use completed projects with code-level fault injection to statistically compare the effects of model-level and code-level faults, similarly to [17], where code-level fault injection was compared to real software faults. Secondly, a specific model checking algorithm could inherently optimize the structure of the state space without lazy evaluation injected into the model (see Section 4.4). This could be further supported with specific programming/modeling languages, for ex- ample by extending C with nondeterministic choice and assignment, or by annotations marking fault injection points in the code. Thirdly, the OSEK case study presented here is only in a preliminary phase – modeling other aspects of the OSEK API and additional error detectors or fault-tolerance mechanisms will be necessary.

References

[1] ISO 17356-1:2005. *Road vehicles – open interface for embedded automotive applications*. Part 1: General structure and terms, definitions and abbreviated terms.

[2] SAE J 1739. *Potential Failure Mode and Effects Analysis in Design (Design FMEA), Potential Failure Mode and Effects Analysis in Manufacturing and Assembly Processes (Process FMEA).* 2009.

[3] Anderson, T., Feng, M., Riddle, S., Romanovsky, A. "Protective wrapper development: A case study." In: Second International Conference, ICCBSS 2003, Ottawa, Canada, Feb. 10–12, 2003. pp. 1-14. https://doi.org/10.1007/3-540-36465-X_1

[4] Arlat, J., Costes, A., Crouzet, Y., Laprie, J. C., Powell, D. "Fault injection and dependability evaluation of fault-tolerant systems." *IEEE Transactions on Computers.* 42(8), pp. 913–923. 1993. https://doi.org/10.1109/12.238482

[5] Avižienis, A., Laprie, J-C., Randell, B. "Fundamental concepts of dependability." Technical report, UCLA CSD Report no. 010028. 2001.

[6] Bernardeschi, C., Fantechi, A., Simoncini, L. "Formal reasoning on fault coverage of fault tolerant techniques: A case study." In: Dependable Computing EDCC-1. First European Dependable Computing Conference Berlin, Germany, Oct. 4–6. 1994. (Echtle, K., Hammer, D., Powell, D. (eds.)). pp. 77–94. Springer, 1994. https://doi.org/10.1007/3-540-58426-9_125

[7] Bernardeschi, C., Fantechi, A., Simoncini, L. "Formally Verifying Fault Tolerant System Designs." *The Computer Journal*. 43(3), pp. 191–205. 2000. https://doi.org/10.1093/comjnl/43.3.191

[8] Bloem, R., Könighofer, B., Könighofer, R., Wang, C. "Shield synthesis: Runtime enforcement for reactive systems." In: International Conference on . Tools and Algorithms for the Construction and Analysis of Systems. 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, Apr. 11-18, 2015. pp. 533–548. https://doi.org/10.1007/978-3-662-46681-0_51

[9] Bonfiglio, V., Montecchi, L., Rossi, F., Lollini, P., Pataricza, A., Bondavalli, A. "Executable models to support automated software FMEA." In: *2015 IEEE 16th International Symposium on High Assurance Systems Engineering (HASE)*. pp. 189–196. 2015. https://doi.org/10.1109/HASE.2015.36

[10] Carreira, J. V., Costa, D., Silva, J. G. "Fault injection spot-checks computer system dependability." *IEEE Spectrum*. 36(8), pp. 50–55. 1999. https://doi.org/10.1109/6.780999

[11] Chen, L., Avizienis, A. "N-version programming: A fault-tolerance approach to reliability of software operation." In*:* Twenty-Fifth International Symposium on Fault-Tolerant Computing 1995, June 27-30, 1995, pp. 113–119. https://doi.org/10.1109/FTCSH.1995.532621

[12] Christmansson, J., Chillarege, R. "Generation of an error set that emulates software faults based on field data." In: FTCS '96 Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96). pp. 304–313. 1996. https://doi.org/10.1109/FTCS.1996.534615

[13] Duraes, J. A., Madeira, H. S. "Emulation of software faults: A field data study and a practical approach." *IEEE Transactions on Software Engineering.* 32(11), pp. 849–867. 2006. https://doi.org/10.1109/TSE.2006.113

[14] Grunske, L., Winter, K., Yatapanage, N., Zafar, S., Lindsay, P. A. "Experience with fault injection experiments for FMEA." *Software: Practice and Experience*. 41(11), pp. 1233–1258. 2011. https://doi.org/10.1002/spe.1039

[15] Holzmann, G. "*The SPIN Model Checker: Primer and Reference Manual*." Addison-Wesley Professional, 2003.

[16] Jia, Y., Harman, M. "An analysis and survey of the development of mutation testing." *IEEE Transactions on Software Engineering*. 37(5), pp. 649–678. 2011.
https://doi.org/10.1109/TSE.2010.62

[17] Madeira, H., Costa, D., Vieira, M. "On the emulation of software faults by software fault injection." In: Proceeding International Conference on Dependable Systems and Networks (DSN), 2000, New York, NY, USA, June 25-28, 2000, pp. 417–426.
https://doi.org/10.1109/ICDSN.2000.857571

[18] Menes, R., Hecht, H. "Software safety and certification: Reintroducing the FMEA." Technical report, SoHaR Incorporated.

[19] Molnár, V., Majzik, I. "Evaluation of fault tolerance mechanisms with model checking." In: Proceedings of the 23rd PhD Mini-Symposium. Budapest University of Technology and Economics, Hungary, Feb. 8-9, 2016, pp. 30-33. 2016.

[20] Moraes, R., Duraes, J., Barbosa, R., Martins, E., Madeira, H. "Experimental risk assessment and comparison using software fault injection." In: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Edinburgh, June 25-28, 2007, pp. 512–521.
https://doi.org/10.1109/DSN.2007.45

[21] Papadopoulos, Y., McDermid, J., Sasse, R., Heiner, G. "Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure." *Reliability Engineering & System Safety*. 71(3), pp. 229–247. 2001.
https://doi.org/10.1016/S0951-8320(00)00076-4

[22] Price, C., Snooke, N. "An automated software FMEA." In: Proceedings of the International System Safety Regional Conference (ISSRC), 2008.

[23] Agrawal, H., Demillo, R. A., Hathaway, B., Hsu, W., Hsu, W., Krauser, E. W., Martin, R. J., Mathur, A. P., Spafford, E. "Design of mutant operators for the C programming language." Technical report. Software Engineering Research Center, Purdue University, 1989.

[24] Voas, E., Charron, F., McGraw, G., Miller, K., Friedman, M. "Predicting how badly "good" software can behave." *IEEE Software*. 14(4), pp. 73–83. 1997.
https://doi.org/10.1109/52.595959

[25] Wallace, M. "Modular architectural representation and analysis of fault propagation and transformation." *Electronic Notes in Theoretical Computer Science*. 141(3), pp. 53–71. 2005.
https://doi.org/10.1016/j.entcs.2005.02.051

[26] Weyuker, E. J. "The Oracle Assumption of Program Testing." In: Proceedings of the 13th International Conference on System Sciences (ICSS), Honolulu, HI, Jan. 1980, pp. 44-49.

[27] Zhang, H., Aoki, T., Chiba, Y. "A SPIN-based approach for checking OSEK/VDX applications." In: *Formal Techniques for Safety-Critical Systems*, Vol. 476, pp. 239–255, Springer, 2015.
https://doi.org/10.1007/978-3-319-17581-2_16